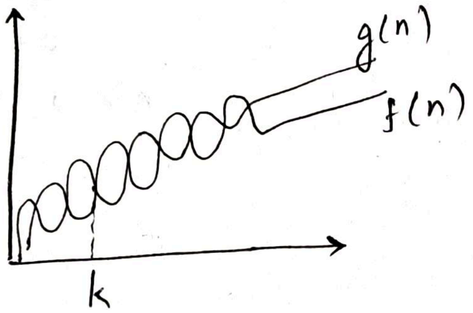
Tutorial Sheet :- 1

Sol 1 :- Asymptotic Notation:- These notations are used to tell the complexity of an algorithm when the i/p is very large.

→ It describes the algorithm efficiency and performance in a meaningful way. It describes the behaviour of time or space complexity for large intance characteristics.

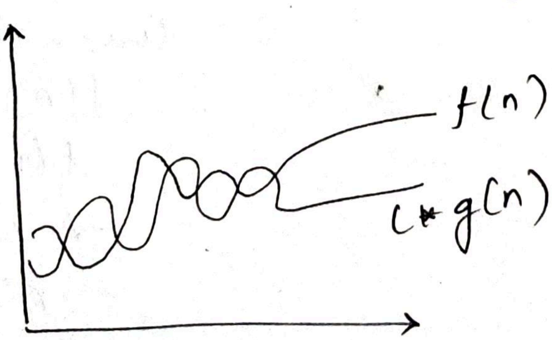
- The asymptotic notation of an algorithm is classified into 5 types :-

(i) **Big Oh notation (O)** :- (Asymptotic upper Bound) The function $f(n) = O(g(n))$, if and only if there exist a +ve constant c and k such that $f(n) \leq c * g(n)$ for all $n \geq k$.



$f(n) = O(g(n))$
iff
$f(n) \leq c \cdot g(n)$
$\forall\ n \geq n_0,$
some constant $c > 0$

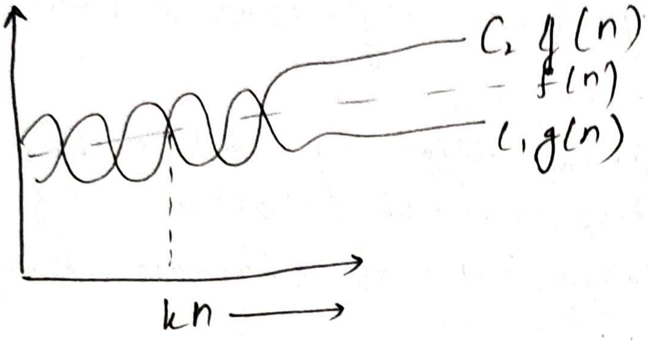
(ii) **Big Omega notation (Ω)** :- (Asymptotic lower bound).
The function $f(n) = \Omega(g(n))$, iff there exits a +ve constant c and k such that $f(n) \geq c * g(n)$ for all $n, n \geq k$.



$f(n) = \Omega\, g(n)$ iff
$f(n) \geq c \cdot g(n)$
$\forall\, n \geq n_0$ & some const $c > 0$

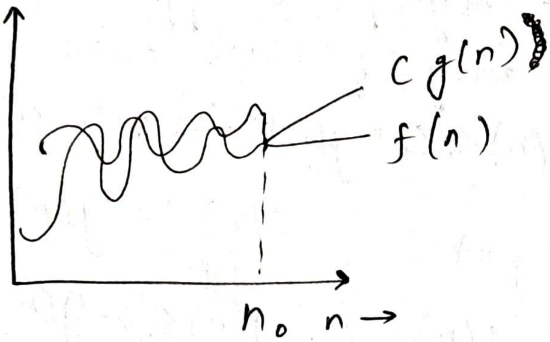
(iii) Big theta notation (θ):- (Asymptotic tight bound):- ②

The function $f(n) = \theta(g(n))$, iff there exits a +ve constant c1, c2, and k such that $c1 * g(n) < f(n) < c2 * g(n)$ for all $n \geq k$.



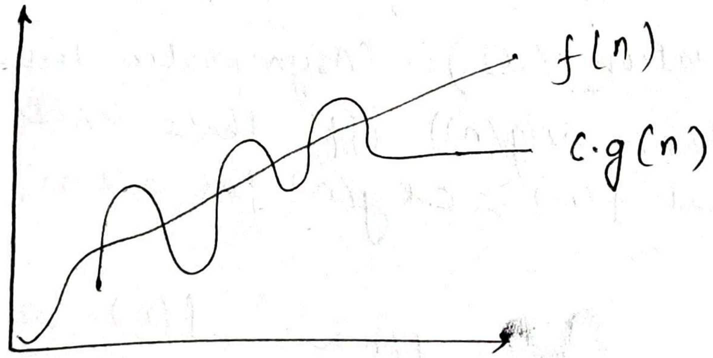
$f(n) = \theta(g(n))$
iff
$C_1 g(n) \leq f(n) \leq C_2 \cdot g(n)$
$\forall\ n \geq \max(n_1, n_2)$

(iv) Small-oh (O) :- O gives us upper bound
$f(n) = o(g(n))$.



$f(n) < c\, g(n)$
$\forall\ n > n_0\ \&\ \forall\ c > 0$
$n = o(n^2)$
$n < 1 n^2$
$2 n^2$
$0 < n^2$
$n < 0.001\, n^2\, n_0$

(v) Small-omega (ω):-



lower bound
$f(n) = \omega\, g(n)$
$f(n) > c \cdot g(n)$
$\forall\ n > n_0\ \&\ \forall\ c > 0$
$n^2 = \omega(n)$

Sol 2 :-

```
for (i=1 to n)
{
    i=i*2;
}
```

Time complexity for a loop means no. of times the loop has to run.

→ For the loop above, the loop will run for following values of i :-

i	1	2	4	8	16	32	64 ----	2^k
value	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^n

$i = 1, 2, 4, 8, 16, 32, \ldots\ldots 2^k$ this means k times

i.e $2^k = n$ [Taking log both sides]

$k \log_2 2 = \log_2 n$

$k = \log_2 n$ $[\because \log_a a = 1]$

$\therefore$ $\boxed{T.C = O(\log n)}$

Sol 3.) $T(n) = \begin{cases} 3T(n-1), & n>0 \\ 1 \end{cases}$

By forward substitution,

$T(n) = 3T(n-1)$

$T(0) = 1$

$T(1) = 3T(1-1)$
$= 3T(0)$
$= 3$

$T(2) = 3T(2-1)$
$= 3T(1)$
$= 3*3 = 3^2$

$T(3) = 3T(3-1)$
$3T(2)$
$= 3*3^2 = 3^3$ .... $T(n) = 3^n$

$\therefore \boxed{TC = O(3^n)}$

Sol 4.) $T(n) = \begin{cases} 2T(n-1) - 1, & n>0 \\ 1 \end{cases}$

By forward substitution

$T(0) = 1$

$T(1) = 2T(n-1) - 1$

$= 2T(1-1) - 1 = 2T(0) - 1$

$= 2 * 1 - 1 = (2-1)$

$T(2) = 2T(2-1) - 1$

$= 2T(1) - 1$

$= 2(2-1) - 1 = 2^2 - 2^1 - 1$

$T(3) = 2T(3-1) - 1$

$= 2T(2) - 1$

$= 2(2^2 - 2^1 - 1) - 1$

$= 2^3 - 2^2 - 2^1 - 1.$

$\vdots$

$T(n) = 2^n - 2^{n-1} - 2^{n-2} - \cdots 2^2 - 2^1 - 2^0$

$= 2^n - (2^n - 1)$

$= \cancel{2^n} - \cancel{2^n} + 1 = 1$

$\therefore \boxed{T(c) = O(1)}$

Sol 5.)

```
int i=1, s=1
while (s<=n)
{
    i++;
    s = s+i;
    printf(" # ");
}
```

$S_i = S_{i-1} + i$

The value of 'i' increases by one for each value contained in 's' at the i^{th} iteration is the sum of the first 'i' +ve integers. If k is the total no. of iterations taken by any program then while loop terminates if: $1+2+3+\dots+k$

$$= [k(k+1)/2] > n$$

So $k = O(\sqrt{n})$

$\therefore$ $\boxed{T.C = O(\sqrt{n})}$

Sol 6.)

```
void function (int n)
{
    int i, count = 0;
    for (i=1; i<=n; i++)      } O(n)
      count++;
}
```

$\boxed{T.C = O(n)}$

Sol 7.)

```
void function (int n)
{
    int i, j, k, count = 0;
    for (i = n/2; i <= n; i++)  → O(n)
        for (j = 1; j <= n; j = j * 2)  → O(log n)
            for (k = 1; k <= n; k = k * 2)  → O(log n)
                count++;
}
```

$T.C = n * \log n * \log n$

$\boxed{T.C = O(n \log^2 n)}$

Sol 8.)

```
function (int n)
{
    if (n==1)
      return;
    for (i=1 to n)            → O(n)
    {
       for (j=1 to n)         → O(n)
       {
           printf ("*");
       }
    }
    function (n-3);
}
```

Time complexity = $O(n^2)$

Sol 9.)

```
void function (int n)
{
    for (i=1 to n) {                  → O(n)
       for (j=1; j<=n; j=j+1)         → O(n)
           printf ("*");
       }
    }
```

$T.C = O(n) * O(n) = \boxed{O(n^2)}$

Sol 10.) n^k is $O(c^n)$ ans

$n^k = O(c^n)$